Equivalence Checking of Hierarchical Combinational Circuits

Williams, Poul Frederick; Hulgaard, Henrik; Andersen, Henrik Reif

Published in:
Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems

Link to article, DOI:
10.1109/ICECS.1999.812296

Publication date:
1999

Document Version
Publisher's PDF, also known as Version of record

EQUIVALENCE CHECKING OF HIERARCHICAL COMBINATIONAL CIRCUITS

Poul Frederick Williams *Henrik Hulgaard* *Henrik Reif Andersen*

Department of Information Technology, Building 344
Technical University of Denmark, DK-2800 Lyngby, Denmark
E-mail: {pfw,henrik,hra}@it.dtu.dk

ABSTRACT

This paper presents a method for verifying that two hierarchical combinational circuits implement the same Boolean functions. The key new feature of the method is its ability to exploit the modularity of the circuits to reuse results obtained from one part of the circuits in other parts. We demonstrate the method on large adder and multiplier circuits.

1 INTRODUCTION

Due to the increase in the complexity of design automation tools and the circuits they manipulate, such tools cannot in general be assumed to be correct. Instead of attempting to formally verify the design automation tools, a more practical approach is to formally check that a circuit generated by a design automation tool functionally corresponds to the original input. This paper presents a technique for formally verifying that two hierarchical combinational circuits implement the same Boolean functions. The presented technique can also be used to check manual modifications of a circuit to ensure that the designer has not introduced errors. Furthermore, the technique can be used to solve sub-problems of other (higher-level) verification problems. For example, verifying arithmetic circuits by checking that they satisfy a given recurrence equation [6] or verifying the equivalence of two state machines without performing a state traversal [16].

In this paper we use a *hierarchical* model of combinational circuits. Based on this model, we show how to propagate a cut through two circuits from the inputs to the outputs. The key new feature of the method is its ability to reuse previously calculated results in the verification. Consider the 4-bit adder in Figure 1. The description consists of two cells; a full-adder cell, *fa*, and a 4-bit adder cell, *4bitadder*, containing four instantiations of the full-adder cell and a description of how they are interconnected. The traditional way of verifying hierarchical combinational circuits is to flatten them into

cell fa(**in** x_0, x_1, x_2; **out** u_0, u_1) {
  $u_0 := x_0$ **xor** x_1 **xor** x_2
  $u_1 :=$ (x_0 **and** x_1) **or** (x_2 **and** (x_0 **or** x_1))
}
cell *4bitadder*(**in** $s_0, \ldots, s_8$;
  out $s_9, s_{11}, s_{13}, s_{15}, s_{16}$) {
  var s_{10}, s_{12}, s_{14}
  instance $fa(s_1, s_2, s_0; s_9, s_{10})$
  instance $fa(s_3, s_4, s_{10}; s_{11}, s_{12})$
  instance $fa(s_5, s_6, s_{12}; s_{13}, s_{14})$
  instance $fa(s_7, s_8, s_{14}; s_{15}, s_{16})$
}

Figure 1: A 4-bit adder consisting of a full-adder cell and a 4-bit adder cell with four instantiations of the full-adder cell.

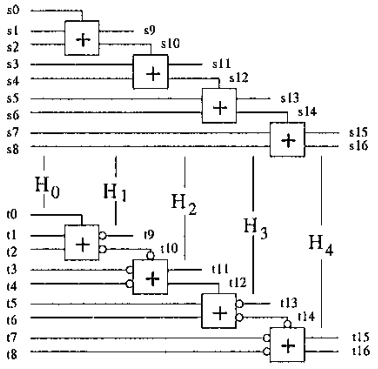

Figure 2: Two 4-bit adders.

a single block of combinational logic on which the verification is performed. In case of complex circuits, this method is not feasible. Our method attempts to work on one cell, and then reuse information about this cell whenever possible.

The 4-bit adder circuit described above corresponds to the top circuit in Figure 2. The bottom circuit in the figure is also a 4-bit adder, but with two instantiations of two different full-adder cells which negate either the inputs or the outputs.

Our method compares the full-adder from the top circuit with each of the two different full-adders in the bottom circuit and combines the results to prove that the two

0-7803-5682-9/99/$10.00©1999 IEEE.

circuits are indeed identical (except for some negated inputs and outputs). The method is automatic as it requires no human interaction during the verification process. If the adders in Figure 2 were larger, our method would still only considers two comparisons between full-adders. The rest of the verification would reuse the comparisons to prove the equivalence.

1.1 Related Work

Ordered Binary Decision Diagrams (OBDDs) [3] is a data structure for representing Boolean functions. A traditional way of verifying two combinational circuits to be equivalent is to build the OBDDs for the circuits and check if the outputs have identical representations. The OBDDs represent the functionality of the circuits, and some functions have no succinct OBDD representation. Therefore it is not always possible to use standard OBDD techniques. Other canonical decision diagrams have been proposed for verification of combinational circuits. One can use other types of decomposition rules [11], relax the variable ordering restriction [7, 9], or extend the domains and/or codomains to integers instead of Booleans [4]. These extensions are typically targeted to solving a particular class of problems.

We have studied the combinational logic-level verification problem for flat circuits [8] using a graph data structure called Boolean Expression Diagram (BED) [1]. This approach works well if the two circuits are similar in structure. However, if the two circuits are very dissimilar in structure, the BED method has the same performance as OBDD methods.

Cerny *et al.* [5] split circuits into cells and each cell is described by a relation between the inputs and the outputs of the cell. Using a sweep strategy, they move either forwards or backwards through the circuits calculating the relations between the circuits along a cut.

Another approach is a structural method which exploits similarities between the two circuits that are compared by identifying related nodes in the circuits and using this information to simplify the verification problem [2, 12]. Such techniques rely on the observation that if two circuits are structurally similar, they will have a large number of internal nodes that are functionally equivalent. Eijk and Janssen [17] use the canonicity of OBDDs to determined whether one node is functionally equivalent to another.

Kunz *et al.* [13] use recursive learning techniques for finding logical implications between nodes in two circuits. Combining learning with OBDDs leads to techniques which can verify larger circuits [15]. The learning technique is further extended by Jain *et al.* [10] and by Matsunaga [14], introducing more general learning methods based on OBDDs and better heuristics for finding cuts in the circuits to split the verification problem into more manageable sizes.

The main differences between all the methods above and our proposed method is that they use a flat circuit description while we use a hierarchical one, and they cannot reuse previously calculated results.

2 Hierarchical Combinational Circuits

Most circuit description languages, for example Berkeley Logic Interchange Format (BLIF), contain language constructs for modular circuit descriptions. Modules may contain other modules yielding a hierarchical description. This leads to a model of *hierarchical combinational circuits* based on cells, instantiations of cells, and connecting wires. There are two types of cells: those that contain instantiations of other cells, *container cells*, and those that contain logic gates, *logic cells*. Definition 1, 2, and 3 define a mathematical model for hierarchical combinational circuits based on these observations.

Definition 1 (HCC) *A* hierarchical combinational circuit *(HCC) is a pair* (C, c)*, where* C *is a set of* cells *and* $c \in C$ *is the top cell.*

For example, Figure 1 describes an HCC (C, c), where $C = \{fa, 4bitadder\}$ and $c = 4bitadder$.

Definition 2 (Cell) *A* cell *c has the following attributes:*

$Vars(c)$, *a set of variables,*

$In(c) \subseteq Vars(c)$, *a list of input variables,*

$Out(c) \subseteq Vars(c)$, *a list of output variables,*

and either

$Inst(c)$, *a list of* instantiations, *or*

$Fct(c)$, *a list of Boolean functions* $In(c) \rightarrow \mathbb{B}$, *one function for each output.*

Container cells have the *Inst* attribute while logic cells have the *Fct* attribute. In the 4-bit adder circuit in Figure 1 there are two cells; the full-adder cell, fa, and the 4-bit adder cell, $4bitadder$. $Vars(fa)$ is the set $\{x_0, x_1, x_2, u_0, u_1\}$. $In(fa)$ is the list $[x_0, x_1, x_2]$, and $Out(fa)$ is the list $[u_0, u_1]$. The full-adder cell has the *Fct* attribute; a list with two elements where the first element is the function for the u_0 output: x_0 **xor** x_1 **xor** x_2. The $4bitadder$ cell has the *Inst* attribute; a list of four instantiations of fa.

Definition 3 (Instantiation) *An* instantiation i *of a cell c has the following attributes:*

$cell(i)$, *the cell c of which i is an instantiation,*

$par(i)$, *the cell in which i is located* $(i \in Inst(par(i)))$,

$in(i) \subseteq Vars(par(i))$, *a list of input variables,*

$out(i) \subseteq Vars(par(i))$, *a list of output variables.*

The instantiation i must further fullfil the requirements:

$|in(i)| = |In(cell(i))|$ *and* $|out(i)| = |Out(cell(i)|$.

The topmost instantiation of a full-adder cell in the 4-bit adder example has $cell(i) = fa, par(i) = 4bitadder$, $in(i) = [s_1, s_2, s_0]$, and $out(i) = [s_9, s_{10}]$.

The outputs of a hierarchical combinational circuit are determined by the inputs. In the case of the 4-bit adder, the outputs are the sum of two 4-bit numbers on the inputs. We use a relation $Rel(c)$ to capture this relation between the inputs and the outputs of a cell c. For a logic cell, Rel is determined by the logic of the gates (the Fct attribute):

$$Rel(c) = \bigwedge_{k=1,\ldots,|Out(c)|} (Out(c)_k \Leftrightarrow Fct(c)_k)\,.$$

(We use characteristic functions to represent relations.) The subscript k indicates the kth element in a list. For example, $Rel(fa)$ is the relation:

$$(u_0 \Leftrightarrow x_0 \oplus x_1 \oplus x_2) \wedge (u_1 \Leftrightarrow (x_0 \wedge x_1) \vee (x_2 \wedge (x_0 \vee x_1)))\,.$$

For container cells, Rel is determined as:

$$Rel(c) = \exists v_{\in V}.\ \bigwedge_{i \in Inst(c)} Rel(cell(i))[Map]\,,$$

where V is the set variables which are neither inputs nor outputs, i.e., $V = Vars(c) \backslash (In(c) \cup Out(c))$, and $[Map]$ is a renaming of $In(cell(i))$ and $Out(cell(i))$ variables to $in(i)$ and $out(i)$ variables, respectively. The notation $\exists v_{\in V}$ for $V = \{v_1, v_2, \ldots, v_k\}$ is shorthand for $\exists v_1.\exists v_2.\cdots.\exists v_k$.

For an HCC (C, c), the relation over the primary inputs and the primary outputs is $Rel(c)$.

We now define a *path* and a *cut* in a cell.

Definition 4 (Path) *For a container cell c, a* path $p = \langle p_1, \ldots, p_n \rangle$ *is a sequence of variables from $Vars(c)$ such that for all k, $1 \leq k < n$, there exists an instantiation $i \in Inst(c)$, such that $p_k \in in(i)$ and $p_{k+1} \in out(i)$.*

Definition 5 (Cut) *A* cut *K in a container cell c is a set of variables from $Vars(c)$ such that any path $p = \langle p_1, \ldots, p_n \rangle$ through c with $p_1 \in In(c)$ and $p_n \in Out(c)$ contains exactly one variable from the cut K.*

For both logic and container cells, the input cut *is the set $In(i)$ and the* output cut *is the set $Out(i)$.*

The set $\{s_3, s_4, \ldots, s_{10}\}$ is a cut in the $4bitadder$ container cell. For two cuts in different HCCs we define a *cut-relation* H as a relation over the values of the variables in the cuts.

Definition 6 (Cut-relation) *A* cut-relation *H between two cuts K_1 and K_2 in two cells is a relation over the values of the variables in K_1 and K_2, i.e., $H \subseteq \mathbb{B}^{K_1 \cup K_2}$.*

A cut-relation over the input cuts of the two circuits in Figure 2 could be:

$$\bigwedge_{i=0,1,2,5,6} (s_i \Leftrightarrow t_i) \wedge \bigwedge_{i=3,4,7,8} (s_i \Leftrightarrow \neg t_i)\,, \qquad (1)$$

stating that s_i and t_i have identical values for $i = 0, 1, 2, 5, 6$, and that s_i and t_i have opposite values for $i = 3, 4, 7, 8$.

We call a cut-relation between input cuts in two cells for an *input relation*. Likewise, we call a cut-relation between output cuts for an *output relation*.

Given a cut-relation H for two cuts K_1 and K_2, and two instantiations i_1 and i_2 such that the input variables of i_1 and i_2 are subsets of K_1 and K_2, respectively, we can determine a relation R_{in} between the input variables of i_1 and i_2:

$$R_{in} = (\exists v_{\in (K_1 \cup K_2) \backslash (in(i_1) \cup in(i_2))}.H)[Map]\,, \qquad (2)$$

where $[Map]$ is a renaming of $in(i_1)$, $in(i_2)$, $out(i_1)$, and $out(i_2)$ variables to $In(cell(i_1))$, $In(cell(i_2))$, $Out(cell(i_1))$, and $Out(cell(i_2))$ variables, respectively.

3 Cut Propagation

Given two hierarchical combinational circuits HCC_1 (C_1, c_1) and HCC_2 (C_2, c_2), and an input relation H_{in}, the verification problem we consider is to determine whether the outputs satisfy a desired relation H_{out}. Typically, H_{in} and H_{out} would represent "the circuits have identical inputs and outputs."

The verification algorithm works by propagating a cut-relation from the inputs to the outputs. Let H_0 be the input relation H_{in}, a cut-relation between the input cuts of c_1 and c_2. We move their cut-relation past instantiations of cells in c_1 and c_2 (assuming that c_1 and c_2 are container cells). In each step we calculate a new cut-relation, H_{k+1}, based on the previous one, H_k. When the cut-relation has reached the outputs, the resulting cut-relation, H_n, relates the outputs of c_1 to the outputs of c_2. If H_n is a subset of H_{out}, $H_n \subseteq H_{out}$, the circuits have the desired output relation.

3.1 Example

Before describing the algorithm in detail, we give an example to illustrate the basic ideas. Consider again the two different implementations of 4-bit adders in Figure 2. The 4-bit adders are described using s and t variables, respectively. The full-adders in the top circuit are described using x (input) and u (output) variables, while the full-adders in the bottom circuit use y and v variables. The H's represent the cut-relations and the vertical lines indicate the cuts.

Assume H_0 in Figure 2 is given by (1). We decide to move the cuts from the inputs to the outputs one full-adder at a time and to move the cuts in the two circuits simultaneously. First we calculate the input relation $R_{in,1}$ between the leftmost full-adder cell in each of the two circuits using (2):

$$R_{in,1} = (x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)\,. \quad (3)$$

Notice the use of cell variables x and y and not instantiation variables s and t, which is important in order to recognize this situation in the future.

Given $R_{in,1}$ and the input/output relation Rel for the two full-adders, we can determine the relation between the outputs (we will show later how to do this):

$$R_{out,1} = (u_0 \Leftrightarrow \neg v_0) \wedge (u_1 \Leftrightarrow \neg v_1)\,. \qquad (4)$$

We move the cuts and determine the new cut-relation H_1 based on $R_{out,1}$ (again, we will show later how to do this):

$$H_1 = \bigwedge_{i=5,6}(s_i \Leftrightarrow t_i) \wedge \bigwedge_{i=3,4,7,8,9,10}(s_i \Leftrightarrow \neg t_i)\,. \qquad (5)$$

In a similar way we propagate the cuts one step further getting H_2:

$$H_2 = \bigwedge_{i=5,6,11,12}(s_i \Leftrightarrow t_i) \wedge \bigwedge_{i=7,8,9}(s_i \Leftrightarrow \neg t_i)\,.$$

In the third step, we start by finding the input relation $R_{in,3}$:

$$R_{in,3} = (x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)\,.$$

This is identical to the relation $R_{in,1}$ from the first step. The full-adders in the third step are also identical to the full-adders in the first step, and thus we can immediately reuse the output relation $R_{out,1}$ instead of calculating $R_{out,3}$. We update the H_2 relation using $R_{out,1}$ and obtain H_3:

$$H_3 = (s_9 \Leftrightarrow \neg t_9) \wedge (s_{11} \Leftrightarrow t_{11}) \wedge (s_{13} \Leftrightarrow \neg t_{13}) \wedge (s_{14} \Leftrightarrow \neg t_{14})\,.$$

Similarly in the fourth step the input relation is found to be identical to that of the second step and the full-adders in the second and the fourth step are identical. We update the relation H_3, and get the final output relation H_4:

$$H_4 = (s_9 \Leftrightarrow \neg t_9) \wedge (s_{11} \Leftrightarrow t_{11}) \wedge (s_{13} \Leftrightarrow \neg t_{13}) \wedge (s_{15} \Leftrightarrow t_{15}) \wedge (s_{16} \Leftrightarrow t_{16})\,.$$

We observe that the sum-bits of the first and third pair of adders have opposite values while the sum-bits of the second and fourth pair of adders are pairwise equivalent.

3.2 Moving Cuts

We distinguish between two ways of moving cuts: `Build` and `Propagate`. `Build` determines the input/output relation Rel for a cell c and uses it to calculate the new cut-relation by moving the cut past an instantiation of cell c. `Propagate` moves cuts past two cells simultaneous by calculating the input relation R_{in} between the inputs of the two cells and from that calculate the output relation R_{out} for the same pair of cells. In the example above we only used `Propagate`.

Algorithm 1 Prop(H, c_1, c_2)

Require: c_1 and c_2 are container cells
1: $(K_1, K_2) \leftarrow$ input cut for (c_1, c_2)
2: **while** K_1, K_2 are not output cuts **do**
3: Select method
4: **if** method is build instantiation $i \in Inst(c_1)$ **then**
5: $(H, K_1) \leftarrow$ `Build`(i, H, K_1)
6: **else if** method is build instantiation $i \in Inst(c_2)$ **then**
7: $(H, K_2) \leftarrow$ `Build`(i, H, K_2)
8: **else if** method is propagate instantiations $i_1 \in Inst(c_1)$ and $i_2 \in Inst(c_2)$ **then**
9: $(H, K_1, K_2) \leftarrow$ `Propagate`(i_1, i_2, H, K_1, K_2)
10: **end if**
11: **end while**
12: **return** H

Algorithm 2 Build(i, H, K)

Require: $in(i) \subseteq K$
1: $Map \leftarrow$ map $In(cell(i))$ to $in(i)$ and $Out(cell(i))$ to $out(i)$
2: $R \leftarrow Rel(cell(i))[Map]$
3: $H' \leftarrow \exists v \in in(i).H \wedge R$
4: $K' \leftarrow K \cup out(i) \backslash in(i)$
5: **return** (H', K')

Algorithm 1 shows the pseudo-code for the overall algorithm `Prop`. The algorithm moves a cut through two container cells by for each step selecting either the `Build` or the `Propagate` algorithm. In the example, `Prop`$(H_0, 4bitadder_1, 4bitadder_2)$ calculates the output relation H_4, where $4bitadder_1$ and $4bitadder_2$ are the two different descriptions of 4-bit adders.

Algorithm 2 shows the pseudo-code for `Build`. It takes three inputs: an instantiation i, a cut-relation H, and a cut K. It is assumed that all input variables for i are in the cut. The lines 1 and 2 calculate the input/output relation for $cell(i)$ using instantiation variables, line 3 calculates the new cut-relation, and line 4 calculates the new cut.

The `Propagate` algorithm shown in Algorithm 3 considers two cell instantiations at a time; one in each circuit. `Propagate` takes five arguments; two instantiations i_1 and i_2, two cuts K_1 and K_2, and a cut-relation H over the cuts. The result is a new cut-relation and two new cuts. It is assumed that the input variables of the cell instantiations i_1 and i_2 belong to the cuts K_1 and K_2, respectively.

In line 1 `Propagate` calculates, using (2), the input relation R_{in} between i_1 and i_2 based on the cut-relation

Algorithm 3 Propagate(i_1, i_2, H, K_1, K_2)

Require: $in(i_1) \subseteq K_1$ and $in(i_2) \subseteq K_2$
1: $R_{in} \leftarrow$ input relation between i_1 and i_2 based on H using (2)
2: **if** memorized $(R_{in}, cell(i_1), cell(i_2))$ **then**
3: $R_{out} \leftarrow$ memorized result
4: **else**
5: either $R_{out} \leftarrow$ `Prop`$(R_{in}, cell(i_1), cell(i_2))$
6: or $R_{out} \leftarrow \exists v_{\in In(cell(i_1)) \cup In(cell(i_2))}.R_{in} \wedge Rel(cell(i_1)) \wedge Rel(cell(i_2))$
7: memorize $(R_{in}, cell(i_1), cell(i_2), R_{out})$
8: **end if**
9: $Map \leftarrow$ map $Out(cell(i_1))$ to $out(i_1)$ and $Out(cell(i_2))$ to $out(i_2)$
10: $H' \leftarrow (\exists v_{\in In(cell(i_1)) \cup In(cell(i_2))}.H) \wedge R_{out}[Map]$
11: $K_1' \leftarrow K_1 \cup out(i_1) \backslash in(i_1)$
12: $K_2' \leftarrow K_2 \cup out(i_2) \backslash in(i_2)$
13: **return** (H', K_1', K_2')

H. The input relation R_{in} is described in cell variables, not in instantiation variables. Next, we calculate the output relation R_{out} for i_1 and i_2. If we have previously propagated a similar cut past the same cells, we reuse the previous result (line 3). Otherwise we have two ways of calculating R_{out}. If both i_1 and i_2 are instantiations of container cells, we can propagate the cut through these instantiations (line 5) by calling `Prop`, which allows us to use `Propagate` on the container cells. Alternatively, we compute R_{out} from the input/output relation Rel for each of the instantiations i_1 and i_2 (line 6). This resembles calling `Build` twice. The rest of the algorithm updates the cuts and calculates the new cut-relation.

In the example, we calculated (4) in line 6 and we calculated the updated cut-relation H_1 (5) in line 10.

3.3 Build vs. Propagate

`Build` works by constructing a representation of the input/output relation for a cell which is used to update the cut-relation H. Such a relation captures the functionality of the cell. Using `Build` on the top cell corresponds to the standard verification method of building the OBDD for the entire circuit. While this works well for smaller circuits, the OBDDs tend to become quite large for more complex circuits.

`Propagate` works by moving a relation between input variables of two cells to a relation between output variables of the same two cells. In case of container cells, `Propagate` moves the cuts one step at a time past instantiations of cells in the container cells. It avoids constructing an OBDD for the functionality of a cell as long as possible. For logic cells it is necessary to construct such an OBDD. However, this OBDD represents only the functionality of a part of the circuit, not the whole circuit, and it is therefore more manageable.

The use of `Propagate` may cause loss of information since it requires contruction of the input relation R_{in} between the cell inputs. Consider the two equivalent circuits in Figure 3. The input cut for the top circuit is $K_1 = \{s_0, s_1\}$ and for the bottom circuit it is $K_2 = \{t_0, t_1\}$. Let H_0 be the cut-relation $(s_0 \Leftrightarrow t_0) \wedge (s_1 \Leftrightarrow t_1)$ when calling `Propagate`. We want to move the cuts past the negation cells. The new cuts contain the variables s_1 and s_2, and t_0 and t_2. We build the input relation R_{in} for the two negation cells:

$$\begin{aligned} R_{in} &= (\exists v_{\in (K_1 \cup K_2) \setminus (in(i_1) \cup in(i_2))}.H_0)[Map] \\ &= (\exists v_{\in \{s_1, t_0\}}.(s_0 \Leftrightarrow t_0) \wedge (s_1 \Leftrightarrow t_1))[Map] \\ &= true\,, \end{aligned}$$

where i_1 and i_2 are instantiations of the two negation cells. In this case R_{in} evaluates to $true$, meaning that the inputs are unrelated; knowing the value of s_0 does not imply a particular calue of t_1, i.e., they are unrelated. $R_{in} = true$ results in H_1 also being $true$ and not the expected $(s_1 \Leftrightarrow \neg t_2) \wedge (\neg s_2 \Leftrightarrow t_0)$.

The problem is that H_0 does not relate the cut variables s_0 and t_1. In general we can state that `Propagate`(i_1, i_2, H, K_1, K_2) works without loss of information if the cut-relation H can be split in two parts: one part containing the variables in $M = in(i_1) \cup in(i_2)$ and one part containing the remaining variables:

$$H \iff (\exists v_{\in M}.H) \wedge (\exists v_{\in (K_1 \cup K_2) \setminus M}.H) \qquad (6)$$

If H can be written as in (6), `Propagate` determines the exact cut-relation H'. Otherwise, it gives a conservative approximation to the output relation $H_{exact} \subseteq H_{approx}$.

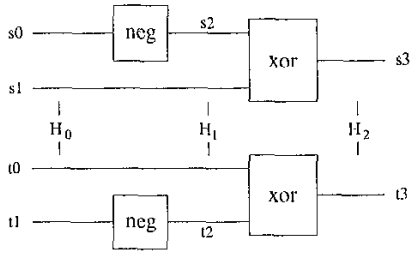

Figure 3: Two combinational circuits. The relation H_0 between the wires in the input cuts is $(s_0 \Leftrightarrow t_0) \wedge (s_1 \Leftrightarrow t_1)$.

4 Experimental Results

To test the proposed method, we have implemented it using OBDDs to represent the characteristic functions of relations. The canonicity allows us to recognize memorized results (line 2 in algorithm 3) in constant time. We have built hierarchical adder and multiplier circuits of different sizes. Each n-bit adder consists of two $n/2$-bit adders. We built one series of adders using the full-adder cells from Figure 1, and another series of adders using two different types of full-adder cells: one full-adder outputting a negated carry-out, and one receiving a negated carry-in signal. The verification task is to verify that given identical inputs, the adders from the two series have identical outputs. The left part of Table 1 shows the runtimes for this experiment. The strategy for moving cuts was to use `Propagate` whenever H can be written as in (6), otherwise we use `Build`. Because of the reuse of previously calculated results, we only apply `Propagate` a number of times proportional to $\log_2(n)$ for n-bit adders.

Using standard OBDD techniques, it is possible to get results comparable to those in Table 1 for the verification of adders since the addition function has a small OBDD representation (when using an appropriate variable order). However, OBDDs are very sensitive to the chosen variable ordering, and using a bad variable order results in OBDDs of size exponential in n making it infeasible to build the OBDDs for the adders. Our proposed method is not sensitive to the variable ordering of the adders as we never build OBDDs representating the functionality of the circuits.

Table 1: Runtimes in seconds on a 500 MHz Digital Alpha to verify pairs of hierarchical adders (left) and pairs of hierarchical multipliers (right) against each other.

Adder [bits]	Runtime [sec]	Multiplier [bits]	Runtime [sec]
64	0.2	32	2.9
128	0.3	64	14.6
256	0.5	128	87.4
512	0.8	256	649
1024	1.6		

We tested the sensitivity to errors of the cut-propagation method by introducing errors into the adders by switching wires around close to the leaves and close to the root in the hierarchy — errors typically arising if wrong parameter lists are given in the circuit descriptions. None of the modifications cause the runtimes to increase significantly.

While adders are easy to handle using OBDDs, multipliers are notoriously difficult. We construct multipliers as series of adders and shifters. From the two different types of adders in the previous experiment, we create two different types of multipliers. The verification task is to verify the pairwise equivalence of outputs given the pairwise equivalence of inputs. One complication is that the outputs of a multiplier are not unrelated. For example, it is not possible for all outputs to be 1 simultaneously [1]. When calculating the cut-relations, such restrictions are included in the relations. This means that the cut-relations contain more information than we need. Repeated use of `Propagate`, even when the cut-relation cannot be written as 6 and thus `Propagate` causes loss of information, turns out to be exactly what is needed to "forget" this extra information. The right part of Table 1 shows the results from running the multiplier experiments.

5 Conclusion

We have presented a method based on cut-propagation for obtaining a relation between the outputs of two hierarchically specified combinational circuits. The key new feature of the method is it ability to exploit the hierarchy in the circuit description to reuse previously calculated results in the verification. We have demonstrated the power of the method by verifying large adders and multipliers.

Acknowledgements

Thanks to Jakob Lichtenberg for suggestions to (6).

[1] For an n-bit multiplier, the greatest result is $(2^n - 1)^2$, which is less than $2^{2n} - 1$, where $2^{2n} - 1$ corresponds to 1 on all outputs.

References

[1] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997.

[2] D. Brand. Verification of large synthesized designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 534–537, 1993.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 535–541, 1995.

[5] E. Cerny and C. Mauras. Tautology checking using cross-controllability and cross-observability relations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1990.

[6] M. Fujita. Verification of arithmetic circuits by comparing two similar circuits. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 159–168. Springer-Verlag, 1996.

[7] J. Gergov and C. Meinel. Efficient Boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Transactions on Computers*, 43(10):1197–1209, Oct. 1994.

[8] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using Boolean expression diagrams. *IEEE Transactions on Computer Aided Design*, July 1999.

[9] J. Jain, J. Bitner, M. S. Abadir, J. A. Abraham, and D. S. Fussell. Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions. *IEEE Transactions on Computers*, 46(11):1230–1245, Nov. 1997.

[10] J. Jain, R. Mukherjee, and M. Fujita. Advanced verification techniques based on learning. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 629–634, 1995.

[11] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Conference on Design Automation (EDAC)*, pages 43–47, 1992.

[12] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, volume 34, pages 263–268, 1997.

[13] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems – test, verification, and optimization. *IEEE Transactions on Computer Aided Design*, 13(9):1143–1158, Sept. 1994.

[14] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 629–634, 1996.

[15] D. K. Pradhan, D. Paul, and M. Chatterjee. VERILAT: Verification using logic augmentation and transformations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 1996.

[16] C. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. International Conf. on Design Automation and Test of Electronic-based Systems (DATE)*, 1998.

[17] C. van Eijk and G. L. J. M. Janssen. Exploiting structural similarities in a BDD-based verification method. In *Theorem Provers in Circuit Design*, number 901 in Lecture Notes in Computer Science, pages 110–125. Springer-Verlag, 1994.